POOMA StyleGuide

version 1.2
January 2, 1996

Notes: Changes from the last version include:

- variable declarations should all occur at the head of a function (makes commenting much easier)
- for statement does not initialize the loop variable within the for statement (g++ get's upset)

1. Control Structure Style

1.1 if statement

For the if statement, place the starting bracket on the same line as the control statement with a space between the right parenthesis and the left bracket. after the keyword if, insert a space before the left parenthesis. Separate the tokens within the if conditional with spaces. Each line within the braces should be indented by two spaces. For nested control structures, the indentation should two times the level of nesting. The closing brace for the if construct is placed on a separate line.

Examples:

```
if ( j < 2) {
  x = 2.0 * y;
  y = 3.0 * z;
}

if ( ( j < 2)  || ( k > 3) ) {
  x = 2.0 * y - gamma * 2.0;
  y = 3.0 * z;
  if ( ( j < 4)  || ( k > 5) ) {
    x += 2.0 * y - gamma * 2.0;
    y = 3.0 * z;
  }
}
```

If the expression is one line and fits on a single line with the if expression, place the expression on the same line as the if keyword and discard the enclosing braces.

Example:

```
if ( j < 2 )  x = gamma * rhos + dt * vParallel + ePhi/tIon;
```

1.2 for statement

For the for statement, place the starting bracket on the same line as the control statement with a space between the right parenthesis and the left bracket. after the keyword for, insert a space before the left parenthesis. Separate the tokens within the for expressions with spaces. Each line within the braces should be indented by two spaces. For nested control structures, the indentation should two times the level of nesting. The closing brace for the for construct is placed on a separate line.
The loop variable for the for loop needs to be initialized before the for loop rather than in the for statement itself. The reason for this is due to non-portability across compilers.

For any given function, initialize all the loop variables at the beginning for the function and add the comment:
For very small routines and functions it is permissable to initialize the loop variable within the for statement. However, in cases where the same loop variable is used more that once in a routine, the loop variables must be declared at the beginning of the function.
// loop variables
Declare the loop variable to be an int within the for expression rather than at a separate point in the code. This will help the user to associate the variable as being a loop variable and is consistent with the philosophy of localizing data declaration with data initialization.

Examples:

```
void MyFunction ( double x, double y, double z ) {
  int j,k; // loop variables

// some code

for ( j = 0 ; j < N ; j++ ) {
  x = 2.0 * y;
  y = 3.0 * z;
}

// more code

for ( j = 0 ; j < N ; j++ ) {
  x = 2.0 * y - gamma * 2.0;
  y = 3.0 * z;
  for ( k = 0 ; k < M ; k++ ) {
    x += 2.0 * y - gamma * 2.0;
    y = 3.0 * z;
  }
}
```

If the expression is one line and fits on a single line with the for expression, place the expression on the same line as the for keyword and discard the enclosing braces.

Example:

```
for ( j = 0 ; j < N ; j++ )  x = gamma * rhos + dt * vParallel ;
```

1.3 while statement

For the while statement, place the starting bracket on the same line as the control statement with a space between the right parenthesis and the left bracket. after the keywordwhile, insert a space before the left parenthesis. Separate the tokens within the while expressions with spaces. Each line within the braces should be indented by two spaces. For nested control structures, the indentation should two times the level of nesting. The closing brace for the while statement is placed on a separate line.

Examples:

```
while ( j < 10 ) {
  x += xstep;
  y += ystep;
}

while ( j < 10 ) {
  x += xstep;
  y += ystep;
```

```
  while ( k < 10 ) {
    x2 += x2step;
    y2 += y2step;
  }
}
```

1.4 switch statement

For the switch statement, place the starting bracket on the same line as the control statement with a space between the right parenthesis and the left bracket. after the keywordswitch, insert a space before the left parenthesis. Switch statements should use enumerations wherever possible to aide in code readability. When placing the enumerator inside the switch statement, allow a space before and after the token between the parentheses.

Within the switch statement, there will be at least one case statement (more likely several...). Eachcase keyword must be indented two spaces in from the column on which the switch keyword begins. The token following thecase keyword must be immediately suffixed by a colon, a space, and then a left brace on the same line. The body of the case statement is to be indented two more spaces and terminated with a break statement.

Finally, everyswitch construct must havedefault as its final case. This allows for code extensibility and is good debugging practice. Thedefault construct is formatted the same way acase construct is formatted with the exception of the keywordcase being replaced bydefault.

The closing braces for theswitch, case, and default constructs are placed on their own line.

```
switch ( color ) {
  case RED: {
     redCount++;
     break;
  }
  case BLUE: {
     blueCount++;
     break;
  }
  default: {
     cout << "no available color" << endl;
     break;
  }
}
```

2. Expression Style

In general, expressions should utilize white space to help bring out the structure of the expression.

Example:

```
t=t1*1.23+t4*1.22-t5*1.26;
```

```
t = t1*1.23 + t4*1.22 - t5*1.26;
```

Here, the second line utilizes spaces to delineate between the addition and multiplication operations, making the expression far more readable.

Example:

```
t=(((2.0*gamma0)+(3.0*gamma1)/b1-1.0)/5);
```

```
t=(( (2.0*gamma0) + (3.0*gamma1)/b1 - 1.0 )/5);
```

Here, the second line utilizes spaces to help delineate the nesting of parentheses.

Although no explicit style will be expounded here, it is recommended that expressions utilize spaces to help delineate the logic contained in blocks of tokens.

3. Variable Declaration and Initialization

All variable should be declared and commented at the beginning of the module in which they are utilized. Where possible, the variables should be initialized on the same line as they are declared. Barring this, the variables should be initialized close to the head of the module where that variable is declared.

Example:

```
void MyClass::MyFunction(int* x, int* y, int num) {
  int i,j,k;       // loop variables
  int counter = 0; // current number of updated values
  int* temp = new int[num]; // temporary array of swapped values
  :
  :
```

Note: on some compilers (such as the T3D), there is a problem with initializing variables inside of a case statement which alleviated by enclosing the case in brackets.

Example:

```
    switch(var){
      case 1:{
        int thisIsLocal;
        thisIsLocal = 1;
        ...
      }
    }
```

4. Variable and Function Naming Conventions

All variable should contain alpha numeric characters with the words being separated by capitalization. Underlines (the "_" character) should be used only for functions, not data variables. Member data variables are distinguished from local variable data by capitalizing the first letter. Non-member data only has the interior words capitalized, not the first letter.

Examples:

MyVar - member data
myVar - local data
numParticles - local data
NumCells - member data
MyReallyUbsurdlyLongVariable - member data
i - local data
hello_world - function

5. Class Design

5.1 Public, Protected, and Private

Each class definition should be placed in a separate .h file (unless the class is a small helper class). The interface definition should consist first of public members, then protected members, then private members.
In terms of formatting, the opening brace for the class definition occurs on the same line as the class keyword. The closing brace for the class definition is on a line by itself. The private, protected, and public keywords must start on the same column as the keyword class.

Example:

class MyClass {

public:

protected:

private:

}

5.2 Requisite Member Functions

All classes should include:

- default constructor (no arguments)
- copy constructor
- destructor
- op=
- op<< (overloaded with a stream)

Depending on class functionality, it is advisable to include

- op==
- op!=
- op>>(overloaded with a stream)

All classes should provide functionality through the member functions which allows the state of an object created with any non-default constructor to be attainable with an object created by the default constructor and invocations of suitable accessor and helper functions.
In the public interface, the constructors and destructors should be declared first, next, all accessor functions should be declared, next the op= function, and next the op<<. The rest of the public interface follows.

Example:

```
class MyClass {

  public:

    // constructor
    MyClass(int i) {
      MyData = i;
    }

    // copy constructor
    MyClass(const MyClass& class) {
      MyData = class.get_MyData();
```

```
    }

    // default  constructor
    MyClass() {};

    // destructor
    ~MyClass() {}

    operator=(const MyClass& class) {
      MyData = class.get_MyData();
    }
    int get_MyData(void)   const { return MyData;}
    void set_MyData(int i)       { MyData = i;}


    friend ostream& operator<<(ostream& out, const MyClass& );
      out << "MyData = " << MyData << endl;
    }

  private:
    int MyData;

};
```

Note that an object created by the first constructor could also be created by a call to the default constructor and then the set_MyData member function.

5.3 member data access

To promote data encapsulation, all member data should be private. It is up to the class designer to determine which data members are accessible through member functions. These member functions to access member data must take a unique form which entails prefixing the actual name of the member data with a get_ , set_ , or return prefix.

- The get_ prefix return the member data value.
- The set_ prefix is a member function which sets the member data's variable.
- The return_ prefix is a member function which return a reference to the member data.

Examples:

Given the integer member data MyData of class MyClass the accessor functions in the class would look like:

```
class MyClass {

  public:

    int  get_MyData     ( void  )  const { return MyData; }
    int& return_MyData  ( void  )        { return MyData; }
    void set_MyData     ( int i )        { MyData =  i; }
    int  get_MyData2    ( void  )  const { return MyData2; }
    int& return_MyData2 ( void  )        { return MyData2; }
    void set_MyData2    ( int i )        { MyData2 = i; }

  private:
    int MyData;
    int MyData2;

};
```

In this part of the class definition please follow the following guidelines:

The "get" member accessor functions should return a const value.
Group the accessor functions in the same area of the class interface.
Order the get, set, and return accessors in the same order that they appear in declared within the class. In the above example, MyData is declared before MyData2 and so the MyData accessor functions are defined before the MyData2 accessor functions in the public interface.

6. Constness

Use wherever possible. Constness prevents you from changing data that isn't supposed to be changed. A member function is const if it doesn't alter member data or any other data it may access. One interesting recommendation was to declare the overloaded assignment operator ("=") as const! This prevents statements like "(a=b)=c" which might be confusing (the a=b part cannot be performed because the result is const and cannot appear on the left-hand side of an assignment operator), and it allows stuff like "a=a" (if the assignment operator was not declared const in addition to the right-hand side argument, this would do a costly const to non-const casting).

7. Preprocessing directives

Every .h file should have a preprocessing directive to check and see if the file has already been included during compilation. These preprocessing directives should take the name of the class in all capital letters followed by an "_H". For example, the class "MyClass" would have a .h file which started with the preprocessing directives

```
#ifndef MYCLASS_H
#define MYCLASS_H

// the class header and definition goes here...

#endif
```

8. Commenting

8.1 Comment Style

In general, comments should focus on the "why" rather than the "how". One wants to avoid the comments merely repeating what the code is already saying. The comments should reflect higher level ideas that can not be directly inferred from the code itself. A reader of a function should be able to read through the comments in the function and obtain a good idea of what the function does.

The Following are general safety tips:

- Keep comments close to the code they describe
- Comment all special uses of the language or environment
- Justify violations of good programming style
- Document non-intuitive regions of code
- Avoid abbreviations
- Delete out-of-date comments, these are deadly
- Comment as you go along, don't wait to do it all when you've forgotten what you wrote!

8.2 Format

The header comments at the beginning of files utilize the /* and */ constructs, everywhere else, the // construct is utilized.

8.3 Major and Minor comments

While commenting your code, distinguish between major and minor descriptions through indentation of the // comments.
For major comments, start commenting at the first column with the // structure.
For minor comments, comment at the current level of code indentation.

Example:

```
   for ( j = 0 ; j < N  ; j++ ) {
     x *= 2.0;
     y *= 4.0:
   }
// Here is a major comment which would probably go
// on for a few lines describing what is to come next.
   for ( j = 0 ; j < N  ; j++ ) {
     r *= 2.0;
     // this would be a minor comment
     if ( j > M) {
       r *= 4.0;
       s *= 4.0:
     }
   }
```

8.4

8.5 Control Structures

Unless completely self evident, every control structure which contains more than a single line of code should have a proceeding comment.
For very long control structures, comments should be placed after the closing right bracket to help readability. Control structures that can be viewed on a single page in a text editor do not require these comments, the structures can be discerned from the code indentation.

Example:

```
// the beginning of Forever
while ( j < doForever ) {

// lot's o'code

} // while ( j < doForever)
```

8.6 Endline comments

Endine comments should be used to clarify a single line of code. Endline comments should not be used for multiple lines of codes since it is difficult to tell which lines are being addressed by the comment.
There are three main uses for endline comments:

- To annotate data declarations
- For code maintenance notes (such as bug fixed)
- Marking the end of long control structures

8.7 Files

8.7.1 .h file

The beginning of every .h file should have a header which includes in the following order:

1. a copyright header
2. the class name
3. an author list
4. a class description
5. version dates

6. a list of parent classes
7. a list of children classes
8. a list of participating classes
9. revisions

The copyright header will delineate code ownership and responsibility in addition to the conditions under which the code my have been co-developed. The exact form of the copyright header is described in the subsection below. The Primary Author for the class should be listed first. Anyone listed as an author for the Class should be familiar with all aspects of its design and implementation.
The version dates should reflect when the class was added to the specified version of the FrameWork. With every new version number of the FrameWork, the date should specify when the classes functionality within the FrameWork is verified.
The parent and children classes reflect the position of the class within the hierarchy of the POOMA FrameWork.
The list of Participating classes describe all classes required by the class in the .h files.
Revisions must be listed with the date, person responsible, and a description of the revision. The revisions should appear in an enumerated list to enable concise referencing. Furthermore, revisions in the .h file should only pertain to changes in the class interface. Changes to the implementation of a given member function should be put into a revision comment in the accompanying .C files.

EXAMPLE:

The below would be a header for the file Stencil.h

```
/////////////////////////////////////////////////////

   (C) Copyright 1996 The Board of Trustees of the
               University of California
                  All Rights Reserved

/////////////////////////////////////////////////////
CLASS:      Stencil

AUTHOR:     John Reynders

VERSION DATES:
  version 1.1 - July 15, 1995
  version 2.1 - November 25, 1995
  version 2.2 - December 25, 1995

DESCRIPTION:
The Stencil class interacts with the field class to
perform finite difference operations. The CEO excepts a
stencil token and points to optimized stencil kernels
registered by the Field class. Stencil objects may
interact with each other through overloaded operators to
produce other Stencil objects. In this manner, complex
stencil operations may be built out of simpler
constitutive operations while retaining a symbolic
representation with efficient implementation via CEO.

PARENT CLASSES:
GData
```

```
CHILDREN CLASSES:
None

PARTICIPATING CLASSES:
ResizeIArray, ResizeDArray, Dfield, CEO, StencilLink

REVISION 1 - 12/3/96 - Jullian Cummings
Added the overloaded operator
  Stencil& operator*(const Stencil&, const Stencil&);

REVISION 2 - 12/6/96 - John Reynders
Added the I/O member functions
  void write_binary(void);
  void write_ascii(void);
and deleted the I/O member function
  void write_now(void);

******************************************************/
```

When the Revisions list gets above 5 items, the list should be placed at the end of the .h file to avoid several page aheads in the editor to get to the class description. In this case the Revisions statement should indicate how many revision exist and that they are located at the end of the file:

EXAMPLE:

The below would be a header for the file Stencil.h

```
/******************************************************

   (C) Copyright 1996 The Board of Trustees of the
               University of California
                  All Rights Reserved

******************************************************
 CLASS:       Stencil

AUTHOR:     John Reynders

VERSION DATES:
  version 1.1 - July 15, 1995
  version 2.1 - November 25, 1995
  version 2.2 - December 25, 1995
  version 2.3 - January 5, 1996

DESCRIPTION:
The Stencil class interacts with the field class to
perform finite difference operations. The CEO excepts a
stencil token and points to optimized stencil kernels
registered by the Field class. Stencil objects may
interact with each other through overloaded operators to
produce other Stencil objects. In this manner, complex
stencil operations may be built out of simpler
constitutive operations while retaining a symbolic
representation with efficient implementation via CEO.

PARENT CLASSES:
GData

CHILDREN CLASSES:
None

PARTICIPATING CLASSES:
```

```
ResizeIArray, ResizeDArray, Dfield, CEO, StencilLink

REVISIONS
There are 9 revision to this class described at the end of
this header file.

*****************************************************/
```

8.7.2 .C file

The beginning of every .C file should have a header which includes in the following order:

1. a copyright header
2. the file name
3. an author list
4. file description
5. date of construction,
6. revisions

The copyright header will delineate code ownership and responsibility in addition to the conditions under which the code my have been co-developed. The exact form of the copyright header is described in the subsection below. The Primary Author for the class should be listed first. Anyone listed as an author for the Class should be familiar with all aspects of its design and implementation.
If the file contains member function to a particular class, the file description should simply describe what member functions of that class are contained within the file (for example - just the I/O routines, just the functions pertaining to particle swap, all member function for the class, etc....)
The Date should reflect when the class was initially added to the FrameWork. Revisions must be listed with the date, person responsible, and a description of the revision. The revisions should appear in an enumerated list to enable concise referencing. Furthermore, revisions in the .C file should not pertain to changes in the class interface (those revision comments should be made in the .h file). The .C file is where changes to the implementation of a given member function should be placed.

EXAMPLE:

The below would be a header for the file StenciIO.C

```
/*******************************************************

   (C) Copyright 1996 The Board of Trustees of the
               University of California
                 All Rights Reserved

*******************************************************
FILE:   StencilIO.C

AUTHOR: John Reynders
DATE:   December 1, 1995

DESCRIPTION:
This file contains the member functions of the Stencil
class responsible for performing I/O operations.
```

```
REVISION 1 - 12/5/96 - John Reynders
Improved efficiency of the overloaded operator
  Stencil& operator*(const Stencil&, const Stencil&);
By unrolling inner loop.

REVISION 2 - 12/8/96 - Julian Cummings
Fixed a bug in the I/O member functions
  void write_binary(void);
which caused the last element of the Stencil to not be
printed.

*****************************************************/
```

As was the case with the header file, when the Revisions list gets above 5 items, the list should be placed at the end of the .C file to avoid several page aheads in the editor to get to the class implementation. In this case the Revisions statement should indicate how many revision exist and that they are located at the end of the file.

8.7.3 The Copyright header:

The copyright header will delineate code ownership and responsibility in addition to the conditions under which the code my have been co-developed. With respect to the current contracts, and grants in place it seems there are eight valid modes of development.

1. The class was developed by LANL
2. The class was developed by NMSU
3. The class was developed by DSS
4. The class was developed by DSS under contract PU-XXXXXXXX.
5. The class was co-developed by DSS and LANL.
6. The class was developed by Parallel Solutions
7. The class was developed by Parallel Solutions under contract PU-XXXXX
8. The class was co-developed by Parallel Solutions and LANL.

For each of these cases, the copyright header should be as follows:

LANL developed:

```
/*****************************************************

        (C) Copyright 1996 The Board of Trustees of the
                    University of California
                        All Rights Reserved

*****************************************************
```

NMSU developed:

```
/*****************************************************

        (C) Copyright 1996 The Board of Trustees of the
                    New Mexico State University
                        All Rights Reserved

*****************************************************
```

DSS developed:

```
/*********************************************************

        (C) Copyright 1996 Dakota Software Systems, Inc.
                    All Rights Reserved

*********************************************************
```

DSS developed under contract.

```
/*********************************************************

        (C) Copyright 1996 The Board of Trustees of the
                    University of California
                    All Rights Reserved

*********************************************************
          Developed for Los Alamos National Laboratory by
        Dakota Software Systems, Inc Under Contract XXXXXX.
*********************************************************
```

DSS and LANL co-developed.

```
/*********************************************************

        (C) Copyright 1996 The Board of Trustees of the
                    University of California
                    All Rights Reserved

*********************************************************
          Co-Developed by Los Alamos National Laboratory and
                    Dakota Software Systems, Inc .
*********************************************************
```

Parallel Solutions developed:

```
/*********************************************************

        (C) Copyright 1996 Parallel Solutions, Inc.
                    All Rights Reserved

*********************************************************
```

Parallel Solutions developed under contract.

```
/*********************************************************

        (C) Copyright 1996 The Board of Trustees of the
                    University of California
                    All Rights Reserved

*********************************************************
```

Developed for Los Alamos National Laboratory by
Parallel Solutions, Inc Under Contract XXXXXX.
****

Parallel Solutions and LANL co-developed.

/**********

****
Co-Developed by Los Alamos National Laboratory and
Parallel Solutions, Inc .
****

8.8 Functions

8.8.1 Description

Directly before the function declaration, the comments should describe the function in a few sentences. If the function can not be described in a few sentences, this is a good clue that the function probably needs to be broken up into smaller functions.

8.8.2 Parameters

Every parameter passed to the function should be listed and defined directly before the function declaration and after the function description. Parameters which act as output variable should preceded those which act as input variables. In defining the parameters, each should be designated as an output or input parameter.

8.8.3 Variables

All variables in a function should be initialized and commented at the beginning of a function. Each variable should have a description of use and where possible an indication of the allowable range of values. Variable declarations should be annotated with endline comments where possible.

8.8.4 Error Correction

Every error found and corrected in a subroutine should be commented. The comment should clearly delineate which part of the code was fixed and reference the revision where the bugfix is described in detail along with the data and the fixer.

8.8.5 Example

```
//-------------------------------------------------
// This function swaps the particle values between
// two different time level using the such-and-such
// method of charge preserving something-or-other.
// levelA (output) : time level 0 particles
// levelB (output) : time level 1 particles
// nox    (input ) : number of particles
// dim    (input ) : Dimension of system
// q      (input ) : the so-and-so charge level [0.0,1.0]
void Particles::LevelSwap(double* levelA, double* levelB,
                          int nox, int dim, double q) {
int i,j,k;         // loop counters
double* tempLevel; // temporary array of particle values
int numCharged;    // # of Charged particles [0,Nox-1]
```

```
// some code

for( j = 0 ; j < Nox ; j++ ) {
  tempLevel[j] = levelA[j];  // BugFix: Revision 9
   :
   :
// some more code

// Begin BugFix: Revision 10
for( j = 0 ; j < Nox ; j++ ) {
  if( MyChargeValue[j] < q ) {
    MyValues[j] = LevelB[j] * q;
  }
}
// End BugFix: Revision 10

// yet even some more code
   :
   :
}
//-----------------------------------------------------
```

8.9 Function Delineation

Between every function use a one line structure of the form:
//----------
to indicate the end of a function and the start of a new one.

Example:

```
//-----------------------------------------
void MyClass::Function1(int j) {
  MyVar = j;
  MyVar2 = 2*j;
}
//-----------------------------------------
void MyClass::Function2(int j) {
  MyVar = 2*j;
  MyVar2 = 4*j;
}
//-----------------------------------------
```

9. Function parameter ordering

Parameters which act as output variable should preceded those which act as input variables. In defining the parameters, each should be designated as an output or input parameter.

10. Enumerations

Enumeratations should have enumerators with all capitalized characters and have the enumeration name as a prefix followed by an underscore and then a unique identifier. The enumeration name should have the first character capitalized. All enumerations should include a default or null enumerator.

Example:

enum Color { COLOR_RED, COLOR_BLUE, COLOR_BLUE, COLOR_NONE };

All enumerated variables should contain alpha numeric characters with the words being separated by capitalization followed by the suffix _E.
Enumerated values are distinguished by the suffix _E and are an exception to the rule that no variable contain the underline ("_") character.

Member enumerated variables are distinguished from local enumerated variables by capitalizing the first character in the variable. Local enumerated variables only have the interior words capitalized, not the first letter.

Examples:

Color MyColor_E = COLOR_BLUE; // initializing a member enumerated variable
Color anotherColor_E = COLOR_RED; // initializing a local enumerated variable

11. Static Variables

All static variables should contain alpha numeric characters with the words being separated by capitalization followed by the suffix _S.
Static variables are distinguished by the suffix _S and are an exception to the rule that no variable contain the underline ("_") character.
Member static variables are distinguished from local static variables by capitalizing the first character in the variable. Local static variables only have the interior words capitalized, not the first letter.
The initialization of all class static variable should occur at the beginning of the source file which contains the constructors for the class. The initialization of all static variable should be proceeded and followed by comments of the form:

```
//////////////////////////// static data ////////////////////////////
```

Examples:

in the .h file we have (ignoring the header comments)

```
class MyClass {

  public:

    // constructor
    MyClass(int i);

    // copy constructor
    MyClass(const MyClass& class);

    // default  constructor
    MyClass() {};

    // destructor
    ~MyClass() {}

    operator=(const MyClass& class);
    int get_MyData(void)   const { return MyData;}
    void set_MyData(int i)       { MyData = i;}


    friend ostream& operator<<(ostream& out, const MyClass& );

  private:
    int MyData;
    static in MyCounter_S;

};
```

in the .C file we have (ignoring the header comments)

#include “MyClass.h”

```
//////////////////////////// static data ////////////////////////////
int MyClass::MyCounter_S = 0;
//////////////////////////// static data ////////////////////////////
// constructor
MyClass::MyClass(int i) {
  MyData = i;
}
//--------------------------------------------------------------------
// copy constructor
MyClass::MyClass(const MyClass& class) {
  MyData = class.get_MyData();
}
//--------------------------------------------------------------------
:
:
```

12. TypeDefs

All static variables should contain alpha numeric characters with the words being separated by capitalization followed by the suffix _T.
All typedef declarations should occur at the head of the .C file in which they are utilized, not in the .h file.

Example .C file:

```
#include "MyClass.h"

typedef int DataType_T;

//-----------------------------------------
void MyClass::Function1(DataType_T j) {
  MyVar = j;
  MyVar2 = 2*j;
}
//-----------------------------------------
void MyClass::Function2(DataType_T j) {
  MyVar = 2*j;
  MyVar2 = 4*j;
}
//-----------------------------------------
```

13. Friends

Where do we declare classes as friends in a class definition. Where do we put the friended functions inside of a class.
All friended classes should be declared before the public section of a class definition.

```
Example:

class MyClass {

friend class MyClass2;

public:

:
:
```

14. Templates, Exceptions, and the STL

Utilization of templates, exception, or the STL is permissible only if the capability is conditionally compiled. The default functionality of all classes within the FrameWork should not depend upon these capabilities since most compilers do not support this extended functionality of

C++. Nonetheless, some compilers (such as the Photon compiler on black) have this functionality. We should start designing our classes to utilize these capabilities but keep their introduction into the framework conditional until the vendors catch up.